1 Objectives

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

In this lecture we also extend the idea of local state to create a simple implementation of objects, and discuss its limitations. We will also show the message dispatch model of objects, which allows for inheritance and virtual functions.

- Be able to explain what an object is.
- Know how to implement an object using records and HOFs.
- Know how to implement an object using a message dispatcher.
- Be able compare the record and dispatcher models.
- Major goal 1: be able to simulate objects in a language lacking them.
- Major goal 2: understand how objects work "under the hood".

2 Examples

Counter, version 1.

```
# let ct = ref 0;;
val ct : int ref = {contents=0}
# let counter () =
    ct := !ct + 1;
    !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Counter, version 2.

```
# let counter =
    let ct = ref 0 in
    fun () -> ct := !ct + 1; !ct;;
val counter : unit -> int = <fun>
```

```
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Why is version 2 okay but version 1 is bad?

3 Problems

Try the following problems. In a few minutes the instructor will go over the solutions. Feel free to work with the person next to you!

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?
2. The random number function generator does not have a way to reset the state. We would also like to be able to ask "what was the last random number generated" without changing the seed. Write a (group of) functions to do this.
3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?

4 Object Examples

What is an object?

- Data and functions are grouped together.
- Functions have their own local state.
- Objects can send and receive *messages*.
- Objects can refer to *themselves*.

Using recursive records.

```
let mkPoint newloc =
  let rec this =
  { loc = ref newloc;
    getx = (fun () -> pi1 !(this.loc));
    gety = (fun () -> pi2 !(this.loc));
    draw = (fun () -> report !(this.loc));
    move = (fun dl ->
              this.loc := movept !(this.loc) dl)}
 in this;;
```

Using a dispatcher.

```
let mkPoint x y =
  let x = ref x in
  let y = ref y in
  fun st ->
```

```
5     match st with
6     | "getx" -> (fun _ -> !x)
7     | "gety" -> (fun _ -> !y)
8     | "movx" -> (fun nx -> x := !x + nx; nx)
9     | "movy" -> (fun ny -> y := !y + ny; ny)
10    |  _ -> raise (Failure "Unknown message.")
```